

Similarity Group-by Operators for Multi-dimensional Relational Data

Mingjie Tang, Ruby Y. Tahboub, Walid G. Aref, *Senior Member, IEEE*,
Mikhail J. Atallah, *Fellow, IEEE and ACM*, Qutaibah M. Malluhi, Mourad Ouzzani, *Member, IEEE*, and
Yasin N. Silva

Abstract—The SQL group-by operator plays an important role in summarizing and aggregating large datasets in a data analytics stack. While the standard group-by operator, which is based on equality, is useful in several applications, allowing similarity aware grouping provides a more realistic view on real-world data that could lead to better insights. The Similarity SQL-based Group-By operator (SGB, for short) extends the semantics of the standard SQL Group-by by grouping data with similar but not necessarily equal values. While existing similarity-based grouping operators efficiently materialize this approximate semantics, they primarily focus on one-dimensional attributes and treat multi-dimensional attributes independently. However, correlated attributes, such as in spatial data, are processed independently, and hence, groups in the multi-dimensional space are not detected properly. To address this problem, we introduce two new SGB operators for multi-dimensional data. The first operator is the clique (or distance-to-all) SGB, where all the tuples in a group are within some distance from each other. The second operator is the distance-to-any SGB, where a tuple belongs to a group if the tuple is within some distance from any other tuple in the group. Since a tuple may satisfy the membership criterion of multiple groups, we introduce three different semantics to deal with such a case: (i) eliminate the tuple, (ii) put the tuple in any one group, and (iii) create a new group for this tuple. We implement and test the new SGB operators and their algorithms inside PostgreSQL. The overhead introduced by these operators proves to be minimal and the execution times are comparable to those of the standard Group-by. The experimental study, based on TPC-H and a social check-in data, demonstrates that the proposed algorithms can achieve up to three orders of magnitude enhancement in performance over baseline methods developed to solve the same problem.

Index Terms—similarity query, relational database,



1 INTRODUCTION

The deluge of data accumulated from sensors, social networks, computational sciences, and location-aware services calls for advanced querying and analytics that are often dependent on efficient aggregation and summarization techniques. The SQL group-by operator is one main construct that is used in conjunction with aggregate operations to cluster the data into groups and produce useful summaries. Grouping is usually performed by aggregating into the same groups tuples with equal values on a certain subset of the attributes. However, many applications (i.e., in Section 5) are often interested in grouping based on *similar* rather than strictly equal values.

Clustering [1] is a well-known technique for grouping similar data items in the multi-dimensional space. In most

cases, clustering is performed outside of the database system. Moving the data outside of the database to perform the clustering and then back into the database for further processing results in a costly impedance mismatch. Moreover, based on the needs of the underlying applications, the output clusters may need to be further processed by SQL to filter out some of the clusters and to perform other SQL operations on the remaining clusters. Hence, it would be greatly beneficial to develop practical and fast similarity group-by operators that can be embedded within SQL to avoid such impedance mismatch and to benefit from the processing power of all the other SQL operators.

SQL-based Similarity Group-by (SGB) operators have been proposed in [2] to support several semantics to group similar but not necessarily equal data. Although several applications can benefit from using existing SGB over Group-by, a key shortcoming of these operators is that they focus on one-dimensional data. Consequently, data can only be approximately grouped based on one attribute at a time.

In this paper, we introduce new similarity-based group-by operators that group multi-dimensional data using various metric distance functions. More specifically, we propose two SGB operators, namely SGB-All and SGB-Any, for grouping multi-dimensional data. SGB-All forms groups such that a tuple or a data item, say o , belongs to a group, say g , if o is at a distance within a user-defined threshold from all other data items in g . In other words, each group in SGB-All forms a clique of nearby

- M. Tang and R. Tahboub is with the Department of Computer Science, Purdue University, Indiana, IN, 47906.
E-mail: tang49@purdue.edu
- W.G. Aref is with the Department of Computer Science, Purdue, and Center for Education and Research in Information Assurance and Security (CERIAS).
- M. Atallah is with the Department of Computer Science, Purdue University.
- Q. Malluhi is with the Department of Computer Science and Engineering, Qatar University.
- M. Ouzzani is with the Qatar Computing Research Institute.
- Y. Silva is with the School of Mathematical and Natural Science, Arizona State University.

data items in the multi-dimensional space. For example, all the two-dimensional points (a-e) in Figure 1a are within distance 3 from each other and hence form a clique. They are all reported as members of one group as they are all part of the output of SGB-All. In contrast, SGB-Any forms groups such that a tuple or a data item, say o , belongs to a group, say g , if o is within a user-defined threshold from at least one other data item in g . For example, all the two dimensional points in Figure 1b form one group. Point a is within Distance 3 from Point c that in turn is within Distance 3 from Points b, d , and f . Furthermore, Point e is within Distance 3 from Point d , and so on. Therefore, Points a-h of Figure 1b are reported as members of one group as part of the output of SGB-Any.

Notice that in the SGB-All operator, a data item may qualify the membership criterion of multiple groups. For example, data item c in Figure 1a forms a clique with two groups. In this case, we propose three semantics, namely, *on-overlap join-any*, *on-overlap eliminate*, and *on-overlap form-new-group*, for handling such a case. We provide efficient algorithms for computing the two proposed SGB operators over correlated multi-dimensional data. The proposed algorithms use a filter-refine paradigm. In the filter step, a fast yet conservative check is performed to identify the data items that are candidates to form groups. Some of the data items resulting from the filter step will end up being false-positives that will be discarded. The refinement step eliminates the false-positives to produce the final output groups. Notice that for the case of SGB-Any, a data item cannot belong to multiple groups. For example, consider a data item, say o , that is a member of two groups, say g_1 and g_2 , i.e., o is within distance ϵ from at least one other data item in each of g_1 and g_2 . In this case, based on the semantics of SGB-Any, Groups g_1 and g_2 merge into one encompassing bigger group that contains all members of g_1 , g_2 and common data item o . Specificity, we mainly focus on two and three dimensional data space, leaving higher dimensions for future work.

The contributions of this paper are as follows:

- 1) We introduce two new operators, namely SGB-All and SGB-Any, for grouping multi-dimensional data from within SQL.
- 2) We present an extensible algorithmic framework to accommodate the various semantics of SGB-All and SGB-Any along with various options to handle overlapping data among groups. We introduce effective optimizations for both operators.
- 3) We prototype the two operators inside PostgreSQL and study their performance using the TPC-H benchmark. The experiments demonstrate that the proposed algorithms can achieve up to three orders of magnitude enhancement in performance over the baseline approaches. Moreover, the performance of the proposed SGB operators is comparable to that of relational Group-by, and outperform state-of-the-art clustering algorithm (i.e., *K-means*, *DBSCAN* and *BIRCH*) from one to three orders of magnitude.

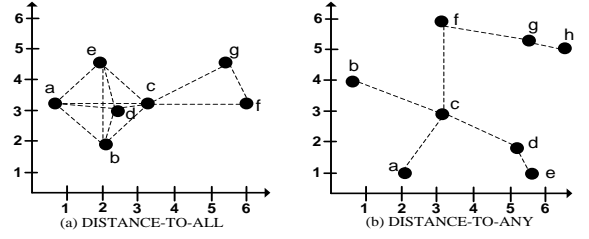


Fig. 1: The Semantics of Similarity predicates $\epsilon = 3$.

The rest of the paper proceeds as follows. Section 2 discusses the related work. Section 3 provides background material. Section 4 introduces the new SGB operators. Section 5 presents application scenarios that demonstrate the use and practicality of the various proposed semantics for SGB operators. Sections 6 and 7 introduce the algorithmic frameworks for SGB-All and SGB-Any operators, respectively. Section 8 describes the in-database extensions to support the two operators and their performance evaluation from within PostgreSQL. Section 9 concludes the paper.

2 RELATED WORK

Previous work on similarity-aware query processing addressed the theoretical foundation and query optimization issues for similarity-aware query operators [2], [3], [4] introduce similarity algebra that extends relational algebra operations, e.g., joins and set operations, with similarity semantics. Similarity queries and their optimizations include algorithms for similarity range search and K-Nearest Neighbor (KNN) [5], similarity join [6], and similarity aggregates [7]. Most of work focus on semantic and transformation rules for query optimization purpose independently from actual algorithms to realize similarity-aware operators. In contrast, our focus is on the latter.

Clustering forms groups of similar data for the purpose of learning hidden knowledge. Clustering methods and algorithms have been extensively studied in the literature, e.g., see [8], [1]. The main clustering methods are partitioning, hierarchical, and density-based. *K-means* [9] is a widely used partitioning algorithm that uses several iterations to refine the output clusters. Hierarchical methods build clusters either divisively (i.e., top-down) such as in *BIRCH* [10], or agglomeratively (i.e., bottom-up) such as in *CURE* [11]. Density-based methods, e.g., *DBSCAN* [12], cluster data based on local criteria, e.g., density reachability among data elements. The key differences between our proposed SGB operators and clustering are: (1) the proposed SGB operators are relational operator that are integrated in a relational query evaluation pipeline with various grouping semantics. Hence, they avoid the impedance mismatch experienced by standalone clustering and data mining packages that mandate extracting the data to be clustered out of the DBMS. (2) In contrast to standalone clustering algorithms, the SGB operators can be interleaved with other relational operators. (3) Standard relational query optimization techniques that apply to the standard relational group-by are also applicable to the SGB operators

as illustrated in [2]. This is not feasible with standalone clustering algorithms. Also, improved performance can be gained by using database access methods that process multi-dimensional data.

An early work on similarity-based grouping appears in [13]. It addresses the inconsistencies and redundancy encountered while integrating information systems with dirty data. However, this work realizes similarity grouping through pairwise comparisons which incur excessive computations in the absence of a proper index. Furthermore, the introduced extensions are not integrated as first class database operators. The work in [14] focuses on overcoming the limitations of the distinct-value group-by operator and introduces the SQL construct “Cluster By” that uses conventional clustering algorithms, e.g., *DBSCAN*, to realize similarity grouping. Cluster By addresses the impedance mismatch due to the data being outside the DBMS to perform clustering. Our SGB operators are more generic as they use a set of predicates and clauses to refine the grouping semantics, e.g., the distance relationships among the data elements that constitute the group and how inter-group overlaps are dealt with.

Several DBMSs have been extended to support similarity operations. SIREN [15] is a similarity retrieval engine that allows executing similarity queries over a relational DBMS. POSTGRESQL-IE [16] is an image handling extension of PostgreSQL to support content-based image retrieval capabilities, e.g., supporting the image data type and responding to image similarity queries. While these extensions incorporate various notions of similarity into query processing, they focus on the similarity search operation. SimDB [2] is a PostgreSQL extension that supports similarity-based queries and their optimizations. Several similarity operations, e.g., join and group-by, are implemented in as first-class database operators. However, the similarity operators in SimDB focus on one-dimensional data and do not handle multi-dimensional attributes.

3 PRELIMINARIES

In this section, we provide background definitions and formally introduce similarity-based group-by operators.

Definition 1: A **metric space** is a space $\mathbf{M} = \langle \mathbb{D}, \delta \rangle$ in which the distance between two data points within a domain \mathbb{D} is defined by a function $\delta : \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{R}$ that satisfies the properties of symmetry, non-negativity, and triangular inequality.

We use the Minkowski distance L_p as the distance function δ . We consider the following two Minkowski distance functions. Let p_x be a data point in the multi-dimensional space of the form $p_x : \langle x_1, \dots, x_d \rangle$ and p_{xy} is the value of the y^{th} dimension of p_x .

- The Euclidean distance

$$L_2 : \delta_2(p_i, p_j) = \sqrt{\sum_y (p_{iy} - p_{jy})^2}$$
- The maximum distance

$$L_\infty : \delta_\infty(p_i, p_j) = \max_y |p_{iy} - p_{jy}|.$$

Definition 2: A **similarity predicate** $\xi_{\delta, \epsilon}$ is a Boolean expression that returns TRUE for two multi-dimensional points, say p_i and p_j , if the distance δ between p_i and p_j is less than or equal to ϵ , i.e., $\xi_{\delta, \epsilon}(p_i, p_j) : \delta(p_i, p_j) \leq \epsilon$. In this case, the two points are said to be similar.

Definition 3: Let T be a relation of tuples, where each tuple, say t , is of the form $t = \{GA_1, \dots, GA_k, NGA_1, \dots, NGA_l\}$, the subset $GA_c = \{GA_1, \dots, GA_k\}$ be the grouping attributes, the subset $NGA = \{NGA_1, \dots, NGA_l\}$ be the non-grouping attributes, and $\xi_{\delta, \epsilon}$ be a similarity predicate. Then, the **similarity Group-by operator** $\mathcal{G}_{(GA_c, (\xi_{\delta, \epsilon}))}(R)$ forms a set of answer groups G_s by applying $\xi_{\delta, \epsilon}$ to the elements of GA_c such that a pair of tuples, say t_i and t_j , are in the same group if $\xi_{\delta, \epsilon}(t_i \cdot GA_c, t_j \cdot GA_c)$.

Definition 4: Given a set of groups $G = \{g_1, \dots, g_m\}$, the **Overlap Set** $Oset$ is the set of tuples formed by the union of the intersections of all pairs of groups (g_1, \dots, g_m) , i.e., $Oset = \cup_{(i,j) \in \{1..m\}} (g_i \cap g_j)$, where $i \neq j$. In other words, $Oset$ contains all the tuples that belong to more than one group.

For simplicity, we study the case where the set of grouping attributes, GA_c , contains only two attributes. In this case, we can view tuples as points in the two-dimensional space, each of the form $p : (x_1, x_2)$. We enclose each group of points by a bounding rectangle $R : (p_l, p_r)$, where points p_l and p_r correspond to the upper-left and bottom-right corners of R , respectively.

4 SIMILARITY GROUP-BY OPERATORS

This section introduces the semantics of the two similarity-based group-by operators, namely, SGB-All and SGB-Any.

4.1 Similarity Group-By ALL (SGB-All)

Given a set of tuples whose grouping attributes form a set, say P , of two-dimensional points, where $P = \{p_1, \dots, p_n\}$, the SGB-All operator \mathcal{G}_{all} forms a set, say G_m , of groups of points from P such that $\forall g \in G_m$, the similarity predicate $\xi_{\delta, \epsilon}$ is TRUE for all pairs of points $\langle p_i, p_j \rangle \in g$, and g is maximal, i.e., there is no group g' such that $g \subseteq g'$. More formally,

$$\mathcal{G}_{all} = \{g \mid \forall p_i, p_j \in g, \xi_{\delta, \epsilon}(p_i, p_j) \wedge g \text{ is maximal}\}$$

Figure 1 gives an example of two groups (a-e) and (c,f,g), where all pairs of elements within each group are within a distance $\epsilon \leq 3$. The proposed SQL syntax for the SGB-All operator is as follows:

```
SELECT column, aggregate-func(column)
FROM table-name
WHERE condition
GROUP BY column DISTANCE-TO-ALL [L2 | LINF] WITHIN  $\epsilon$ 
ON-OVERLAP [JOIN-ANY | ELIMINATE | FORM-NEW-GROUP]
```

SGB-All uses the following clauses to realize similarity-based grouping:

- **DISTANCE-TO-ALL:** specifies the distance function to be applied by the similarity predicate when deciding the membership of points within a group.

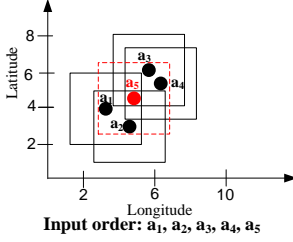


Fig. 2: Data points using $\epsilon = 3$ and L_∞ .

- L2: L_2 (Euclidean distance).
- LINF: L_∞ (Maximum infinity distance)
- ON-OVERLAP: is an arbitration clause to decide on a course of action when a data point is within Distance ϵ from more than one group. When a point, say p_i , matches the membership criterion for more than one group, say $g_1 \dots g_w$, one of the three following actions are taken:
 - JOIN-ANY: the data point p_i is randomly inserted into any one group out of $g_1 \dots g_w$.
 - ELIMINATE: discard the data point p_i , i.e., all data points in $Oset$ (see Definition 4) are eliminated.
 - FORM-NEW-GROUP: insert p_i into a separate group, i.e., form new groups out of the points in $Oset$.

Example 1: The following query performs the aggregate operation *count* on the groups formed by SGB-All on the two-dimensional grouping attributes *GPSCoor-lat* and *GPSCoor-long*. The L_∞ distance is used with Threshold $\epsilon = 3$.

```
SELECT count(*)
FROM GPSPoints
GROUP BY GPSCoor-lat, GPSCoor-long DISTANCE-TO-ALL LINF
WITHIN 3
ON-OVERLAP <clause>
```

Consider Points a_1 - a_5 in Figure 2 that arrive in the order a_1, a_2, \dots, a_5 . After processing a_4 , the following groups satisfy the SGB-All predicates: $g_1 \{a_1, a_2\}$ and $g_2 \{a_3, a_4\}$. However, Data-point a_5 is within ϵ from a_1, a_2 in g_1 and a_3, a_4 in g_2 . Consequently, an arbitration ON-OVERLAP clause is necessary. We consider the three possible semantics below for illustration.

With an ON-OVERLAP JOIN-ANY clause, a group is selected at random. If g_1 is selected, the resulting groups are $g_1 \{a_1, a_2, a_5\}$ and $g_2 \{a_3, a_4\}$, and the answer to the query is $\{3, 2\}$. With an ON-OVERLAP ELIMINATE clause, the overlapping point a_5 gets dropped; the resulting groups are $g_1 \{a_1, a_2\}$ and $g_2 \{a_3, a_4\}$, and the query output is $\{2, 2\}$. With an ON-OVERLAP FORM-NEW-GROUP clause, the overlapping point a_5 is inserted into a newly created group; the resulting groups are $g_1 \{a_1, a_2\}$, $g_2 \{a_3, a_4\}$, $g_3 \{a_5\}$ and the query output is $\{2, 2, 1\}$.

4.2 Similarity Group-By Any (SGB-Any)

Given a set of tuples whose grouping attributes from a set, say P , of two dimensional points, where $P = \{p_1, \dots, p_n\}$,

the SGB-Any operator \check{G}_{any} clusters points in P into a set of groups, say G_m , such that, for each group $g \in G_m$, the points in g are all connected by edges to form a graph, where an edge connects two points, say p_i and p_j , in the graph if they are within Distance ϵ from each other, i.e., $\xi_{\delta, \epsilon}(p_i, p_j)$. More formally,

$$\check{G}_{any} = \{g \mid \forall p_i, p_j \in g, (\xi_{\delta, \epsilon}(p_i, p_j) \vee (\exists p_{k1}, \dots, p_{kn}, \xi_{\delta, \epsilon}(p_i, p_{k1}) \wedge \dots \wedge \xi_{\delta, \epsilon}(p_{kn}, p_j))) \wedge g \text{ is maximal}\}$$

The notion of distance-to-any between elements within a group is illustrated in Figure 1b, where $\epsilon = 3$. All of the points (a-h) form one group. The corresponding SQL syntax of the SGB-Any operator is as follows:

```
SELECT column, aggregate-func(column)
FROM table-name
WHERE condition
GROUP BY column DISTANCE-TO-ANY [L2 | LINF] WITHIN epsilon
```

SGB-Any uses the DISTANCE-TO-ANY predicate that applies the metric space function while evaluating the distance between adjacent points. When using the semantics for SGB-Any, the case for points overlapping multiple groups does not arise. The reason is that when an input point overlaps multiple groups, the groups merge to form one large group.

Example 2: The following query performs the aggregate operation *count* on the groups formed by SGB-Any on the two-dimensional grouping attributes *GPSCoor-lat* and *GPSCoor-long* using the Euclidean distance with $\epsilon = 3$.

```
SELECT count(*)
FROM GPSPoints
GROUP BY GPSCoor-lat and GPSCoor-long
DISTANCE-TO-ANY L2 WITHIN 3
```

Consider the example in Figure 2. After processing a_4 , the following groups are $g_1 \{a_1, a_2\}$ and $g_2 \{a_3, a_4\}$. Since Point a_5 is within ϵ from both a_1, a_2 in g_1 and a_3, a_4 in g_2 , the two groups are merged into a single group. Therefore, the output of the query is $\{5\}$. Any overlapping point will cause groups to merge and hence there is no need to add a special clause to handle overlaps.

5 APPLICATIONS

In this section, we present application scenarios that demonstrate the practicality and the use of the various semantics for the proposed Similarity Group-by operators.

Example 3: Mobile Ad hoc Network (MANET) is a self-configuring wireless network of mobile devices (e.g., personal digital assistants). A mobile device in a MANET communicates directly with other devices that are within the range of the device's radio signal or indirectly with distant mobile devices using gateways (i.e., intermediate mobile devices, e.g., m_1 and m_2 in Figure 3a). In a MANET, wireless links among nearby devices are established by broadcasting special messages. Radio signals are likely to overlap. As a result, uncaredful broadcasting may result in redundant messages, contention, and collision on communication channels. Consider the Mobile Devices table in

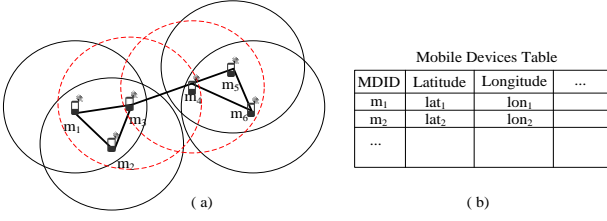


Fig. 3: (a) An Mobile Ad hoc Network (MANET) containing the devices $m_1 \dots m_6$, where the circle around each device is its signal range, (b) The corresponding Mobile Devices table.

Figure 3b that maintains the geographic locations of the mobile devices in a MANET. In the following, we give example queries that illustrate how MANETs can tremendously benefit from SGB-All and SGB-Any operators.

Query 1: Geographic areas that encompass a MANET. A mobile device, say m , belongs to a MANET if and only if m is within the *signal range* from at least one other device mobile. The SGB-ANY semantics identifies a connected group of mobile devices using *signal range* as a similarity grouping threshold.

```
SELECT ST_Polygon(Device-lat, Device-long)
FROM MobileDevices
GROUP BY Device-lat, Device-long
DISTANCE-TO-ANY L2 WITHIN SignalRange
```

Referring to the mobile devices in Figure 3a, the output of Query 1 returns a polygon that encompasses mobile devices m_1 - m_6 .

Query 2: Candidate gateway mobile devices. A gateway represents an overlapping mobile device that connects two devices that are not within each other's signal range. The SGB-All FORM-NEW-GROUP inserts the overlapped devices into a new group. Therefore, those devices in the newly formed group are ideal gateway candidates.

```
SELECT COUNT(*)
FROM MobileDevices
GROUP BY Device-lat, Device-long
DISTANCE-TO-ALL L2 WITHIN SignalRange
ON-OVERLAP FORM-NEW-GROUP
```

The output of Query 2 returns the number of candidate gateway mobile devices. Along the same line, identifying mobile devices that cannot serve as a gateway is equally important to a MANET. SGB-All ELIMINATE identifies mobile devices that cannot serve as a gateway by discarding the overlapping mobile devices.

Example 4: Location-based group recommendation in mobile social media. Several social mobile applications, e.g., *WhatsApp* and *Line*, employ the frequent geographical location of users to form groups that members may like to join. For instance, users who reside in a common area (e.g., within a distance threshold) may share similar interests and are inclined to share news. However, members who overlap several groups may disclose information from one group to another and undermine the privacy of the overlapping groups. Query 3 demonstrates how SGB-ALL allows forming

location-based groups without compromising privacy.

Query 3: Forming private location-based groups. The various SGB-All semantics form groups while handling ON-OVERLAP options that restrict members to join multiple groups. In Query 3, we assume that Table *Users-Frequent-Location* maintains the users' data, e.g., user-id and frequent location. The user-defined aggregate function *List-ID* returns a list that contains all the user-ids within a group.

```
SELECT List-ID(user-id),
ST_Polygon(User-lat, User-long)
FROM Users - Frequent - Location
GROUP BY User-lat, User-long
DISTANCE-TO-ALL L2 WITHIN Threshold
[ON-OVERLAP JOIN-ANY | ELIMINATE | FORM-NEW-GROUP]
```

The output of Query 3 returns a list of user-ids for each formed group along with a polygon that encompasses the group's geographical location. The JOIN-ANY semantics recommends any one arbitrary group for overlapping members who in this case will not be able to join multiple groups. The ELIMINATE semantics drops overlapping members from recommendation, while FORM-NEW-GROUP creates dedicated groups for overlapping members.

6 ALGORITHMS FOR SGB-ALL

In this section, we present an extensible algorithmic framework to realize similarity-based grouping using the distance-to-all semantics with the various options to handle the overlapping data among the groups.

6.1 Framework

Procedure 1 illustrates a generic algorithm to realize SGB-All. This generic algorithm supports the various data overlap semantics using one algorithmic framework. The algorithm breaks down the SGB-All operator into procedures that can be optimized independently. For each data point, the algorithm starts by identifying two sets (Line 2). The first set, namely *CandidateGroups*, consists of groups that p_i can join. p_i can join a group, say g , in *CandidateGroups* if the similarity predicate is true for all pairs $\langle p_i, p'_i \rangle \forall p'_i \in g$. The second set, namely *OverlapGroups*, includes groups that have some (but not all) of its data points satisfying the similarity predicate. A group, say g , is in *OverlapGroups* if there exist at least two points p and q in g such that the similarity distance between p_i and p holds and the similarity distance between p_i and q does not hold. *OverlapGroups* serves as a preprocessing step required to handle the semantics of ELIMINATE and FORM-NEW-GROUP encountered in later steps. Figure 4 gives four existing groups g_1 - g_4 while Data-point x is being processed. In this case, *CandidateGroups* contains $\{g_2, g_3\}$ and *OverlapGroups* contains $\{g_1\}$.

Procedure *ProcessGroupingALL* (Line 3 of Procedure 1) uses *CandidateGroups* and the ON-OVERLAP clause *CLS* to either (i) place p_i into a new group, (ii) place p_i into existing group(s), or (iii) drop p_i from the output,

Procedure 1: Similarity Group-By ALL Framework

Input: P : set of data points, ϵ : similarity threshold, δ : distance function, CLS : ON-OVERLAP clause, G : set of existing groups

Output: Set of output groups

```

1 for each data element  $p_i$  in  $P$  do
2    $(CandidateGroups, OverlapGroups) \leftarrow$ 
      $FindCloseGroups(p_i, G, \epsilon, \delta, CLS)$ 
3    $ProcessGroupingALL(p_i, CandidateGroups, CLS)$ 
4   if  $CLS$  is not JOIN-ANY And  $sizeof(OverlapGroups) \neq 0$  then
5      $ProcessOverlap(p_i, OverlapGroups, CLS)$ 
6   end
7 end

```

in case of an ON-OVERLAP clause. Finally, Procedure *ProcessOverlap* (Line 5) uses *OverlapGroups* to verify whether additional processing is needed to fulfill the semantics of SGB-All.

6.2 Finding Candidate and Overlap Groups

In this section, we present a straightforward approach to identify *CandidateGroups* and *OverlapGroups*. In Section 6.3, we propose a new two-phase filter-refine approach that utilizes a conservative check in the filter phase to efficiently identify the member groups in *CandidateGroups*. Then, in Section 6.4, we introduce the refine phase that is applied only if L_2 is used as the distance metric to detect the *CandidateGroups* that falsely pass the filter step.

Procedure 2 gives the pseudocode for *Naive FindCloseGroups* that evaluates the distance-to-all similarity predicate between p_i and all the points that have been previously processed (Lines 6-15). The grouping semantics (Lines 16-20) identify how the two sets *CandidateGroups* and *OverlapGroups* are populated.

6.2.1 Processing New Points

The second step of the SGB-All Algorithm in Procedure 1 places p_i , the data point being processed, into a new group or into an existing group, or drops p_i from the output depending on the semantics of SGB-All specified in the query.

Procedure 3 (*ProcessGroupingAll*) proceeds as follows. First, it identifies the cases where *CandidateGroups* is empty or consists of a single group. In these cases, p_i is inserted into a newly created group or into an existing group depending on p_i 's distance from the existing group. Procedure *ProcessInsert* places the data point p_i into a group. Next, the ON-OVERLAP clause CLS is consulted to determine the proper course of action. The JOIN-ANY clause arbitrates among the overlapping groups by inserting p_i into a randomly chosen group. The procedure *ProcessEliminate* (Line 13) handles the details of processing the ELIMINATE clause. Consider the example illustrated in Figure 4, where *CandidateGroups* consists of $\{g_2, g_3\}$. *ProcessEliminate* drops Point x .

Finally, Procedure *ProcessNewGroup* (Line 15) processes the FORM-NEW-GROUP clause. It inserts p_i into a

Procedure 2: Naive FindCloseGroupsALL

Input: p_i : data point, ϵ : similarity threshold, δ : distance function, CLS : ON-OVERLAP clause, G : set of existing groups

Output: *Candidate*, *OverlapGroups*

```

1  $Candidate \leftarrow NULL$ 
2  $OverlapGroups \leftarrow NULL$ 
3 for each group  $g_j$  in  $G$  do
4    $CandidateFlag = True$ 
5    $OverlapFlag = False$ 
6   for each  $p_k$  in  $g_j$  do
7     if  $(Distance(p_i, p_k, \delta) \leq \epsilon)$  then
8        $OverlapFlag = True$ 
9     else
10       $CandidateFlag = False$ 
11      if  $CLS == JOIN-ANY$  then
12        break
13    end
14  end
15 end
16 if  $CandidateFlag$  is True then
17   insert  $g_j$  into  $Candidate$ 
18 else if  $CLS$  is not JOIN-ANY and  $CandidateFlag$  is False and  $OverlapFlag$  is True then
19   insert  $g_j$  into  $OverlapGroups$ 
20 end
21 end

```

Procedure 3: ProcessGroupingALL

Input: p_i : data point, CLS : ON-OVERLAP clause, *CandidateGroups*

Output: updates *CandidateGroups* based on CLS semantics

```

1 if  $sizeof(CandidateGroups) == 0$  then
2   create a new group  $g_{new}$ 
3    $ProcessInsert(p_i, g_{new})$ 
4 else if  $sizeof(CandidateGroups) == 1$  then
5   insert into existing group  $g_{out}$ 
6    $ProcessInsert(p_i, g_{out})$ 
7 else
8   switch  $CLS$  do
9     case JOIN-ANY
10       $g_{out} \leftarrow$ 
         $GetRandomGroup(CandidateGroups)$ 
         $ProcessInsert(p_i, g_{out})$ 
11    case ELIMINATE
12       $ProcessEliminate(p_i, CandidateGroups)$ 
13    case FORM-NEW-GROUP
14       $ProcessNewGroup(p_i, CandidateGroups)$ 
15    endsw
16  end
17 end
18 end

```

temporary set termed S' for further processing. The SGB-All with FORM-NEW-GROUP option forms groups out of S' by calling SGB-All recursively until S' is empty.

6.2.2 Handling Overlapped Points

The final step of SGB-All in Procedure 1 processes the groups in the Set *OverlapGroups*. *OverlapGroups* consists of groups, where each group has some data points (but not all of them) that satisfy the similarity predicate with the new input point p_i . This step is required by the

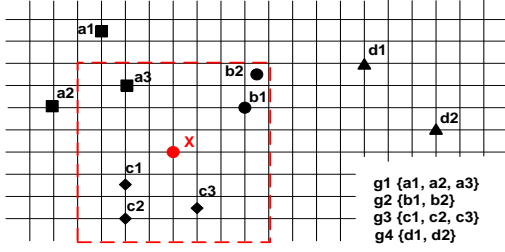


Fig. 4: Processing the point x using L_∞ with $\epsilon = 4$.

ELIMINATE and FORM-NEW-GROUP semantics. Procedure *ProcessOverlap* handles the ELIMINATE semantics as follows. It iterates over *OverlapGroups* and deletes overlapped data points. Consider the example illustrated in Figure 4. Set *OverlapGroups* consists of $\{g_1\}$ with overlapped Data-Point a_3 . Finally, *ProcessOverlap* handles the FORM-NEW-GROUP semantics by inserting the overlapped data points into a temporary set termed S' and deletes these points from their original groups.

The time complexity for SGB-All according the algorithmic framework in Procedure 1 is dominated by the time complexity of *FindCloseGroups*. The time complexity of *ProcessGrouping* and *ProcessOverlap* (Lines 3-6) is linear in the size of *CandidateGroups* and *OverlapGroups*. Consequently, given an input set of size n , Procedure *Naive FindCloseGroups* incurs $\binom{n}{2}$ distance computations that makes the upper-bound time complexity of SGB-All quadratic i.e., $O(n^2)$. Section 6.3 introduces a filter-refine paradigm to optimize over Procedure *Naive FindCloseGroups*.

6.3 The Bounds-Checking Approach

In this section, we introduce a Bounds-Checking approach to optimize over Procedure *Naive FindCloseGroups*. Consider the data points of Group g illustrated in Figure 5a. Procedure *Naive FindCloseGroups* performs six distance computations to determine whether a new data point x can join Group g . To reduce the number of comparisons, we introduce a bounding rectangle for each Group g in conjunction with the similarity threshold ϵ so that all data points that are bounded by the rectangle satisfy the distance-to-all similarity predicate. For example, Data Element x in Figure 5b is located inside g 's bounding rectangle. Therefore, g is a candidate group for x .

Definition 5: Given a set of multi-dimensional points and a similarity predicate $\xi_{\delta_\infty, \epsilon}$, the ϵ -All Bounding Rectangle $R_{\epsilon-All}$ is a bounding rectangle such that for any two points x_i and y_i bounded by $R_{\epsilon-All}$, the similarity predicate $\xi_{\delta_\infty, \epsilon}(x_i, y_i)$ is true.

Consider Figure 5c, where the bounding rectangle $R_{\epsilon-All}$ is constructed for a group that consists of a single Point a_1 , where $\epsilon = 2$ and the sides of the rectangle are 2ϵ by 2ϵ centered at a_1 . After inserting the second Point a_2 into g , as in Figure 5d, $R_{\epsilon-All}$ is shrunk to include the area where the similarity predicate is true for both Points a_1 and a_2 . The invariant that $R_{\epsilon-All}$ maintains varies depending

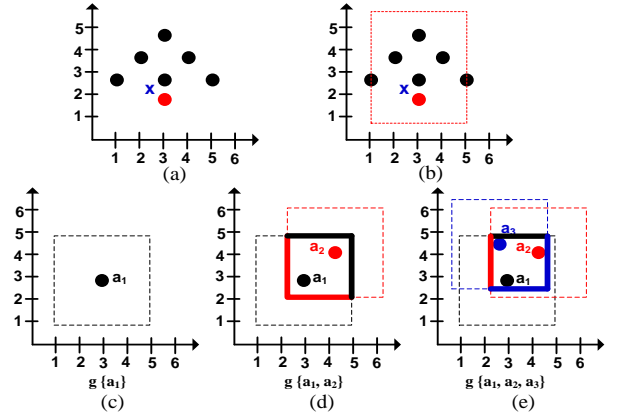


Fig. 5: The ϵ -All Bounding Rectangle approach.

on the distance metric used. For the L_∞ distance metric, $R_{\epsilon-All}$ is updated such that if a Point, say x_i , is inside $R_{\epsilon-All}$, then x_i is guaranteed to be within Distance ϵ from all the points that form Group g . For the Euclidean distance, the invariant that $R_{\epsilon-All}$ maintains is that if a point, says x_i , is outside $R_{\epsilon-All}$, then x_i cannot belong to Group g . In this case, if x_i is inside $R_{\epsilon-All}$, it is likely that x_i is within distance ϵ from all the points inside $R_{\epsilon-All}$. Hence, for the Euclidean distance, $R_{\epsilon-All}$ is a conservative representation of the group g and serves as a filter step to save needless comparisons for points that end up being outside of the group. We illustrate in Figures 5c- 5e how to maintain these invariants when a new point joins the group. We use the case of L_∞ for illustration. When a new point x_i is inside the bounding rectangle $R_{\epsilon-All}$ of Group g , then x_i is within Distance ϵ from all the points in the group, and hence will join Group g . Once x_i joins Group g , the bounds of Rectangle $R_{\epsilon-All}$ are updated to retain the truth of $R_{\epsilon-All}$'s invariant. The sides of $R_{\epsilon-All}$ will need to shrink and will be updated as illustrated in Figures 5d-5e.

Notice that deciding membership of a point into the group requires a constant number of comparisons regardless of the number of points inside Group g . Furthermore, the maintenance of the bounding rectangle of the group takes constant time for every inserted point into g . Also, notice that $R_{\epsilon-All}$ stops shrinking if its dimensions reach $\epsilon \times \epsilon$, which is a lower-bound on the size of $R_{\epsilon-All}$. Figure 5e gives the updated $R_{\epsilon-All}$ after Point a_3 is inserted into the group.

Procedure 4 gives the pseudocode for *Bounds-Checking FindCloseGroups*. The procedure uses the ϵ -All bounding rectangle to reduce the number of distance computations needed to realize *FindCloseGroups* using the L_∞ distance metric. Procedure *PointInRectangleTest* (Line 4) uses the ϵ -All rectangle to determine in constant time whether g_j is a candidate group for the input point. Procedure *OverlapRectangleTest* (Line 6) tests whether the ϵ -All rectangle of p_i overlaps Group g_j 's bounding rectangle. In case of an overlap, all data points in g_j are inspected to verify whether the overlap is nonempty. The correctness of the ϵ -All bounding rectangle for the L_∞ distance metric

Procedure 4: Bounds-Checking FindCloseGroups

Input: p_i : data point, ϵ : similarity threshold, δ : distance function, CLS : ON-OVERLAP clause, G : set of existing groups

Output: *CandidateGroups*, *OverlapGroups*

```

1 CandidateGroups  $\leftarrow$  NULL
2 OverlapGroups  $\leftarrow$  NULL
3 for each group  $g_j$  in  $G$  do
4   if PointInRectangleTest( $p_i, g_j$ ) is True then
5     insert  $g_j$  into CandidateGroups
6   else if  $CLS$  is not JOIN-ANY and
     OverlapRectangleTest( $p_i, g_j$ ) is True then
7     for each  $p_k$  in  $g_j$  do
8       if ( $\text{Distance}(p_i, p_k, \delta) \leq \epsilon$ ) then
9         insert  $g_j$  into OverlapGroups
10        break
11      end
12    end
13  end
14 end

```

Procedure 5: Index Bounds-Checking FindCloseGroups

Input: p_i : data point, ϵ : similarity threshold, δ : distance function, CLS : ON-OVERLAP clause, G : set of existing groups, *Groups_IX*: index on G 's bounding rectangles

Output: *CandidateGroups*, *OverlapGroups*

```

1 CandidateGroups  $\leftarrow$  NULL
2 OverlapGroups  $\leftarrow$  NULL
3  $R_{p_i} \leftarrow \text{CreateBoundingRectangle}(p_i, \epsilon)$ 
4  $GSet \leftarrow \text{WindowQuery}(p_i, R_{p_i}, \text{Groups\_IX})$ 
5 for each group  $g_j$  in  $GSet$  do
6   if PointInRectangleTest( $p_i, g_j$ ) is True then
7     insert  $g_j$  into CandidateGroups
8   else if  $CLS$  is not JOIN-ANY then
9     for each  $p_k$  in  $g_j$  do
10      if ( $\text{Distance}(p_i, p_k, \delta) \leq \epsilon$ ) then
11        insert  $g_j$  into OverlapGroups
12        break
13      end
14    end
15  end
16 end

```

follows from the fact that the rectangles are closed under intersection, i.e., the intersection of two rectangles is also a rectangle.

A major bottleneck of the bounding rectangles approach is in the need to linearly scan all existing bounding rectangles that represent the groups to identify sets *CandidateGroups* and *OverlapGroups*, which is costly. To speedup Procedure *Bounds-Checking FindCloseGroups*, we use a spatial access method (e.g., an R-tree [17]), to index the $R_{\epsilon-All}$ bounding rectangles of the existing groups.

Procedure 5 gives the pseudocode for *Index Bounds-Checking FindCloseGroups*. The procedure performs a window query on the index *Groups_IX* (Line 4) to retrieve the set $GSet$ of all groups that intersect the bounding rectangle R_{p_i} for the newly inserted point p_i . Next, it iterates over $GSet$ (Lines 4-11) and executes *PointInRectangleTest* to determine whether the inspected group belongs to either one of the two sets *CandidateGroups* or *OverlapGroups*. Finally, the elements of *OverlapGroups* are inspected to retrieve the subset of elements that satisfy the similarity predicate.

Refer to Figure 6 for illustration. An R-tree index, termed *Groups_IX*, is used to index the bounding rectangles of the groups discovered so far. In this case, *Groups_IX* contains bounding rectangles for Groups g_1 - g_4 . Given the newly arriving Point x , a window query of the ϵ -All rectangle for x is performed on *Groups_IX* that returns all the intersecting rectangles; in this case, g_1 , g_2 , and g_3 . The outcome of the query is used to construct the sets *CandidateGroups* and *OverlapGroups*.

6.4 Handling False Positives L_2

In this section, we study the effect of using L_2 as a similarity distance function on the SGB-All operator. Refer to Figure 7a for illustration. In contrast to the L_∞ distance, the set of points that are exactly ϵ away from a_1 in the L_2 metric space form a circle. Inserting a_2 (Figure 7b) is

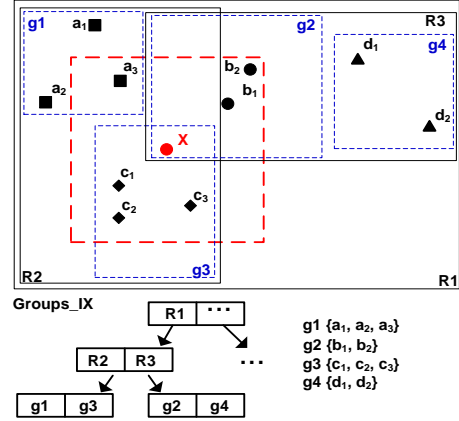


Fig. 6: SGB-All: performing a window Query on *Groups_IX* using $\epsilon = 4$ and L_∞

correct using the L_∞ distance since a_2 is inside the ϵ -All rectangle of a_1 's group. However, under the L_2 distance, a_2 is more than ϵ away from a_1 since a_2 lies outside a_1 's ϵ -circle. As a result, all points that are inside a_1 's ϵ -All group rectangle but are outside the ϵ -circle (i.e., the grey-shaded area in Figure 7b) falsely pass the bounding rectangle test.

Procedure *Naive FindCloseGroups* in (Procedure 2) inspects all input data points. Therefore, the problem of false-positive points does not occur. On the other hand, the Bounds-Checking approach introduced in Procedures 4 and 5 uses the ϵ -All rectangle technique to identify the sets *CandidateGroups* and *OverlapGroups* and hence must address the issue of false-positive points for the L_2 distance metric.

We introduce a **Convex Hull Test** to refine the data points that pass the Bounds-Checking filter step. Given a group of points, a convex hull [18] is the smallest convex set of points within a group. In Figure 7c, the points a_1 - a_5 form the convex hull set for Group g . Based on the SGB-All semantics, the diameter of the convex hull (i.e., the

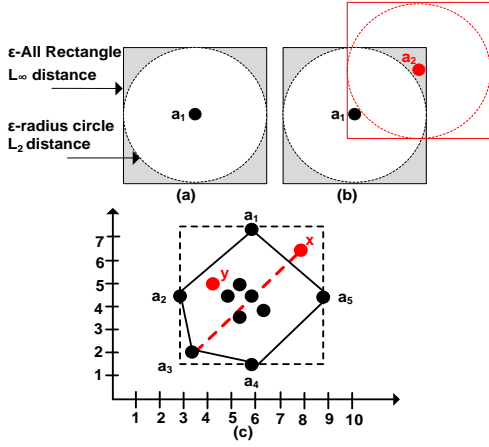


Fig. 7: (a) The ϵ -radius circle in L_2 , (b) The problem of false positive for L_2 , (c) The ϵ -convex hull test for $\epsilon = 6$.

Procedure 6: Convex Hull Test

Input: p_i : data point, g : existing group
Output: True if p_i is not false positive, False otherwise

```

1 ConvexHullSet  $\leftarrow$  getConvexHull( $g$ )
2 if  $p_i$  inside convex hull then
3   return True
4 else
5   farthestPoint  $\leftarrow$ 
     getMaxDistElem(ConvexHullSet,  $p_i$ )
6   if distance(farthestPoint,  $p_i$ )  $\leq \epsilon$  then
7     return True
8   end
9 end
10 return False
```

two farthest points) satisfies the similarity predicate.

The *Convex Hull Test*, illustrated in Procedure 6, verifies whether a point is a false-positive. This additional test can be inserted immediately after (Line 4) in Procedure 4 or immediately after (Line 6) in Procedure 5. Consequently, any new point that lies inside a group's convex hull (e.g., Point y in Figure 7c) satisfies the similarity predicate. In addition, in order to verify points that are outside the convex hull (e.g., Point x in Figure 7c), it is enough to evaluate the similarity predicate between p_i and the convex hull. The correctness of the convex hull test follows from the fact that the convex hull set contains the farthest point from p_i , say p_f . Therefore, it is sufficient to evaluate the similarity predicate between p_i and p_f (e.g., Point x and Point a_3 in Figure 7c). Section 8.1 discusses the complexity of the convex hull approach.

7 ALGORITHMS FOR SGB-ANY

In this section, we present an algorithmic framework to realize similarity-based grouping using the distance-to-any semantics. The generic SGB-Any framework in Procedure 7 proceeds as follows. For each data point, say p_i , Procedure *FindCandidateGroups* (Line 2) uses the distance-to-any similarity predicate to identify the set *CandidateGroups* that consists of all the existing groups that p_i can join. In contrast to SGB-All, in the distance-to-any semantics,

Procedure 7: Similarity Group-By ANY Framework

Input: P : set of data points, ϵ : similarity threshold, δ : distance function, *Points_IX*: spatial index
Output: Set of groups G

```

1 for each data element  $p_i$  in  $P$  do
2   CandidateGroups  $\leftarrow$ 
     FindCandidateGroups( $p_i$ , Points_IX,  $\epsilon$ ,  $\delta$ )
3   ProcessGroupingANY( $p_i$ , CandidateGroups)
4 end
```

a point, say p_i , can join a candidate group, say g , when p_i is within a predefined similarity threshold from at least one another point in g . Procedure *ProcessGroupingANY* (Line 3) inserts p_i into a new or an existing group.

7.1 Finding Candidate Groups

A Naive *FindCandidateGroups* approach similar to Procedure 2 can identify the set *CandidateGroups*. However, this solution incurs many distance computations, and brings the upper-bound time complexity of the SGB-Any framework to $O(n^2)$. The filter-refine paradigm using an ϵ -group bounds-checking approach while applying a distance-to-any predicate (i.e., similar to Procedures 4-6) suffers from two main challenges. By drawing squares of size $\epsilon \times \epsilon$ around the input point and forming a bounding rectangle that encloses all these squares results in a consecutive chain-like region and the area of false-positive progressively increases in size as we add new data points. Furthermore, the convex hull approach to test for false-positive points cannot be applied in SGB-Any as it suffers from false-negatives caused by the fact that the length of the diameter of the convex hull can actually be more than ϵ in the case of SGB-Any. Details are omitted here for brevity.

Consequently, *FindCandidateGroups* in Procedure 8 uses an R-tree index, termed *Points_IX*. *Points_IX* maintains the previously processed data points to efficiently find *CandidateGroups*. Refer to Figure 8 for illustration. For an incoming point, say Point x , an ϵ -rectangle (Line 2 of Procedure 8) is created to perform a window query on *Points_IX* to retrieve *PointsSet* (Line 3). *PointsSet* corresponds to the points that are within ϵ from x , e.g., $\{a_3, c_1, c_2, c_3, b_1, b_2\}$. Based on the semantics of SGB-Any, *CandidateGroups* contains the groups that cover the points in *PointsSet*. For instance, point a_3 belongs to g_1 , points $\{c_1-c_3\}$ belong to g_2 , and points $\{b_1-b_2\}$ belong to group g_3 . Hence, *CandidateGroups* = $\{g_1, g_2, g_3\}$. Procedure *GetGroups* (Line 7) employs a Union-Find data structure [19] to keep track of existing, newly created, and merged groups (see Figure 8b) to efficiently construct *CandidateGroups* given *PointsSet*.

7.2 Processing New Points

Procedure 9 gives the pseudocode for *ProcessGroupingANY*. Lines 1-6 identify the cases when *CandidateGroups* is empty, or when it consists of one group. In these cases, p_i is inserted into a newly created

Procedure 8: FindCandidateGroups

Input: p_i : data point, $Points_IX$: spatial index, δ : distance function, ϵ : similarity threshold
Output: $CandidateGroups$

- 1 $CandidateGroups \leftarrow NULL$
- 2 $R_{p_i} \leftarrow CreateBoundingRectangle(p_i, \epsilon)$
- 3 $PointsSet \leftarrow WindowQuery(p_i, R_{p_i}, Points_IX)$
- 4 **if** δ is L_2 **then**
- 5 $PointsSet \leftarrow VerifyPoints(Points_IX, \delta, \epsilon)$
- 6 **end**
- 7 $CandidateGroups \leftarrow GetGroups(PointsSet)$
- 8 insert p_i into $Points_IX$

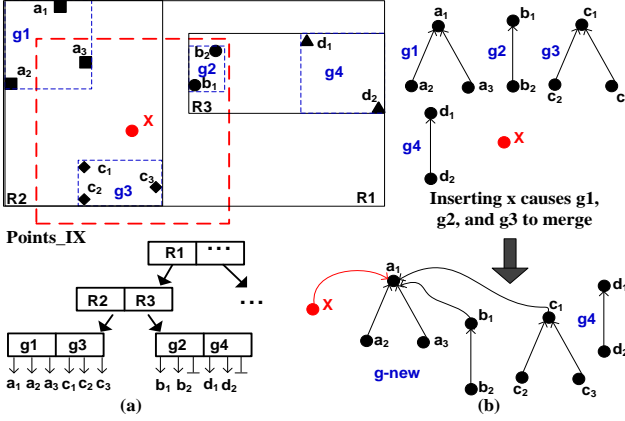


Fig. 8: (a) SGB-Any: Performing a window query on $Points_IX \epsilon = 4$ using L_∞ (b) The disjoint data structure Union-Find is used to maintain existing groups.

group or into an existing group. Next, it handles the case that occurs when p_i is close to more than one group. In the SGB-Any semantics, all candidate groups that p_i can join are merged into one group. Therefore, Procedure *MergeGroupsInsert* (Line 8) handles merging candidate groups and then inserts p_i into the merged groups. Referring to Figure 8b, Point x overlaps groups g_1, g_2 , and g_3 . Based on the semantics of SGB-Any, the overlapped groups g_1, g_2 , and g_3 are merged into one encompassing bigger group, termed G_{new} . In this case, the root pointers of g_1, g_2 and x in the Union-Find data structure are redirected to Point a_1 .

8 COMPLEXITY, REALIZATION, AND EVALUATION

8.1 Complexity Analysis

Table 1 summarizes the average-case running time of SGB-All using the proposed optimizations for the L_∞ distance metric. The *All-Pairs* algorithm corresponds to naive *FindCloseGroups* in Procedure 1. Similarly, *Bounds-Checking* and *On-the-fly Indexing* corresponds to the *Bounds-Checking* and *Index Bounds-Checking* optimizations, where $|G|$ is the number of output Groups and m is the recursion depth for the ON-OVERLAP FORM-NEW. In addition, the average-case running time of SGB-Any when

Procedure 9: ProcessGroupingANY

Input: p_i : data point, $CandidateGroups$
Output: updates $CandidateGroups$

- 1 **if** $CandidateGroups$ is Empty **then**
- 2 create a new group g_{new}
- 3 $ProcessInsert(p_i, g_{new})$
- 4 **else if** $sizeof(CandidateGroups) == 1$ **then**
- 5 insert into existing group g_{out}
- 6 $ProcessInsert(p_i, g_{out})$
- 7 **else**
- 8 $MergeGroupsInsert(CandidateGroups, p_i)$
- 9 **end**

using the index is $O(n \log n)$. The worst-case and best-case running times, and detailed analysis are given in the Appendix.

	JOIN-ANY	ELIMINATE	FORM-NEW-GROUP
All-Pairs	$O(n^2)$	$O(n^2)$	$O(n^3)$
Bounds-Checking	$O(n G)$	$O(n G)$	$O(mn G)$
on-the-fly Index	$O(n \log G)$	$O(n \log G)$	$O(mn \log G)$

TABLE 1: SGB-All Complexity for the L_∞ distance

Business Question: Retrieve large volume customers	
GB1	Same as the TPC-H-Q18
Business Question: Retrieve customers with similar buying power, account balance	
SGB1 or SGB2	SELECT max(ab), min(tb), max(tb), average(ab), array_agg(R1.c_custkey) FROM (SELECT c_custkey, c_acctbal as ab FROM Customer WHERE c_acctbal > 100) as R1 (SELECT o_custkey, sum(o_totalprice) as tp FROM Orders, Lineitem WHERE o_orderkey in (SELECT l_orderkey FROM lineitem GROUP BY R1_orderkey having sum(l_quantity) > 3000) and o_orderkey = l_orderkey and o_totalprice > 30000) as R2 WHERE R1.c_custkey = R2.o_custkey GROUP BY ab, tp DISTANCE-ALL WITHIN ϵ USING lone/ltwo on_overlap join-any/form-new/eliminate or GROUP BY ab, tp DISTANCE-ANY WITHIN ϵ USING lone/ltwo
Business Question: Report profit on a given line of parts (by supplier nation and year)	
GB2	Same as the TPC-H-Q9
Business Question: Report profit and shipment time of parts share similar profit and shipment date	
SGB3 or SGB4	SELECT count(), sum(tprof), sum(stime) FROM (SELECT ps_partkey as partkey, sum(l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity) as tprof, sum(l_receiptdate - l_shipdate) as stime FROM lineitem, partsupp, supplier WHERE ps_partkey = l_partkey and s_suppkey = ps_suppkey GROUP BY ps_partkey) as profit GROUP BY tprof, stime DISTANCE-ALL WITHIN ϵ on_overlap join-any/form-new/eliminate or GROUP BY tprof, stime DISTANCE-ANY WITHIN ϵ USING lone/ltwo
Business Question: Determines top supplier who contributed the most to the overall revenue for parts)	
GB3	Same as the TPC-H-Q15
Business Question: Report supplier who contributed the similar profit and account balance	
SGB5 or SGB6	SELECT array_agg(s_suppkey), sum(r.revenue), sum(s_acctbal) FROM (SELECT l_suppkey as supkey, sum(l_extendedprice * (1 - l_discount)) as trevenue, sum(s_acctbal) As acctbal FROM Lineitem WHERE l_shipdate > date '1995-01-01' and l_shipdate < date '1996-01-01' + interval '10' month GROUP BY l_suppkey) as r GROUP BY r.trevenue, s_acctbal DISTANCE-ALL WITHIN ϵ USING lone/ltwo on_overlap join-any/form-new/eliminate or GROUP BY r.trevenue, s_acctbal DISTANCE-ANY WITHIN ϵ USING lone/ltwo

TABLE 2: Performance Evaluation Queries on TPC-H

8.2 Implementation

We realize the proposed SGB operators inside PostgreSQL. In the *parser*, the grammar rules, and actions related to

the “SELECT” statement syntax are updated with similarity keywords (e.g., DISTANCE-TO-ALL and DISTANCE-TO-ANY) to support the SGB query syntax. The parse and query trees are augmented with parameters that contain the similarity semantics (e.g., the threshold value and the overlap action). The *Planner and Optimizer* routines use the extended query-tree to create a similarity-aware plan-tree. In this extension, the optimizer is manipulated to choose a hash-based SGB plan.

The executor modifies the hash-based aggregate group-by routine. Typically, an aggregate operation is carried out by the incremental evaluation of the aggregate function on the processed data. However, the semantics of ON-OVERLAP ELIMINATE and ON-OVERLAP FORM-NEW-GROUP can realize final groupings only after processing the complete dataset. Therefore, the aggregate hash table keeps track of the existing groups in the following way. First, the aggregate hash table entry (AggHashEntry) is extended with a TupleStore data structure that serves as a temporary storage for the previously processed data points. Next, referring to the Bounds-Checking FindCloseGroups presented in Procedure 4, each group’s bounding rectangle is mapped into an entry inside the hash directory. Bounds-Checking FindCloseGroups linearly iterates over the hash table directory to build the sets *CandidateGroups* and *OverlapGroups*. The Index Bounds-Checking in Procedure 5 employs a spatial index to efficiently look up all existing groups a data point can join. Consequently, we extend the executor with an in-memory R-tree that efficiently indexes the existing groups’ bounding rectangles.

In the implementation of FindCloseGroupsAny in Procedure 8, a spatial index is created to maintain the set of points that have been processed and assigned to groups. Moreover, we extend the executor with the Union-Find data structure Disjoint-set forest to support the operations *GetGroups* and *MergeGroupsInsert*.

8.3 Datasets

The goal of the experimental study is to validate the effectiveness of the proposed SGB-All and SGB-Any operators using the optimization methods discussed in Sections 6 and 7. The datasets used in the experiments are based on the TPC-H benchmark¹ [20], and two real-world social checking datasets, namely Brightkite² and Gowalla³ [21]. Table 2 shows the queries used for performance evaluation experiments on TPC-H data. The multi-dimensional attribute is the combination of different tables. For example, SGB queries, i.e., SGB1/SGB2, are combination of Customer and Order Table, and the number of tuples in the Customer and Order tables is $150K \times SF$ and $1500K \times SF$, respectively, where the scale factor SF ranges from 1 to 60. For Brightkite and Gowalla data, SGB queries follow Queries 1 and 3 to cluster users into

groups by the corresponding users’ check-in information (i.e., latitude and longitude).

The experiments are performed on an Intel(R) Xeon (R) E5320 1.86 GHz 4-core processor with 8G memory running Linux, and using the default configuration parameters in PostgreSQL. At first, we focus on the time taken by SGB and hence disregard the data preprocessing time, (e.g., the inner join and filter predicates in Query 18). Furthermore, to understand the overhead of new SGB query, we calculate SGB response time with complicated queries (e.g., the SGB Query 3 to 6). In the paper, we only give the execution time of the L_2 distance metric because the performance when using the L_∞ distance metric exhibits a similar behavior.

8.4 Effect of similarity threshold ϵ

The effect of the similarity threshold ϵ on the query runtime is given in Figure 9 for SGB-Any and all three overlap variants of SGB-All; JOIN-ANY, ELIMINATE and FORM-NEW-GROUP. The experimental data consists of 0.5 million records. The similarity threshold ϵ varies from 0.1 to 0.9.

Consider an unskewed dataset, performing SGB-All using a smaller value of ϵ (e.g., 0.1 or 0.2) forms too many output groups because the similarity predicate evaluates to true on small groups of the data. Increasing the value of ϵ forms large groups that decreases the expected number of output groups. Thus, we observe in Figure 9a, 9b, 9c that the runtime of SGB-All using the various semantics decreases as the value of ϵ approaches 0.9 with the exception of ϵ of value 0.7. The slight increase in runtime in the JOIN-ANY and FORM-NEW-GROUP semantics can be attributed to the distribution of the experimental data.

The runtime and speedup in Figure 9a, 9b, 9c validate the advantage of the optimizations for *Bounds-Checking* and *on-the-fly Index* over *All-Pairs*. The *on-the-fly Index* approach shows two orders of magnitude speedup over *All-Pairs*, and *Bounds-Checking* approach wins one order magnitude faster than that of *All-Pairs*. The reason is that *All-Pairs* realizes similarity grouping by inspecting all pairs of data points in the input, and its runtime is bounded by the input size. In contrast, *Bounds-Checking* defines group bounds in conjunction with the similarity threshold to avoid excessive runtime while grouping. Therefore, the runtime of *Bounds-Checking* is bounded by the number of output groups. Lastly, indexing output groups using *on-the-fly Index* alleviates the effect of the number of output groups on the overall runtime and makes it steady across the various ON-OVERLAP options.

The effect of the similarity threshold ϵ on the query runtime for the SGB-Any query is given in Figure 9d. The experiment illustrates that the runtime for *All-Pairs* SGB-Any decreases as the value of ϵ increases. Furthermore, the runtime of the *on-the-fly Index* method slightly changes. As a result, the speedup between the *All Pairs* and the *on-the-fly Index* methods slightly decreases. The runtime result validates that the performance of the *on-the-fly Index* method is stable as we vary the value of ϵ . The reason

1. <http://www.tpc.org/tpch/>

2. <https://snap.stanford.edu/data/loc-brightkite.html>

3. <https://snap.stanford.edu/data/loc-gowalla.html>

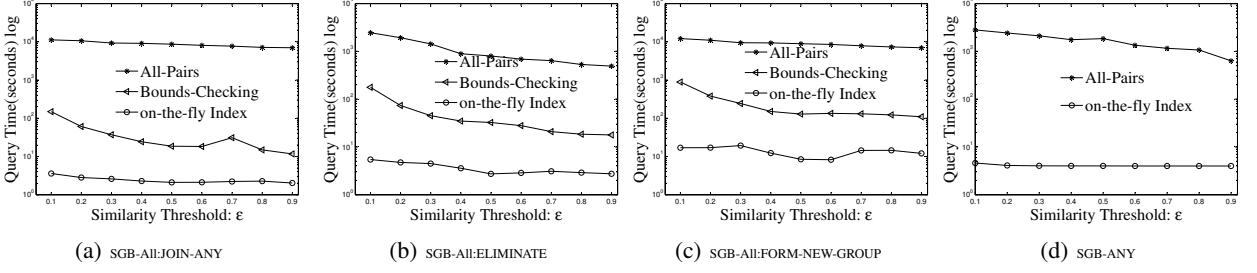


Fig. 9: The effect of similarity threshold ϵ on the SGB-All variants and SGB-ANY

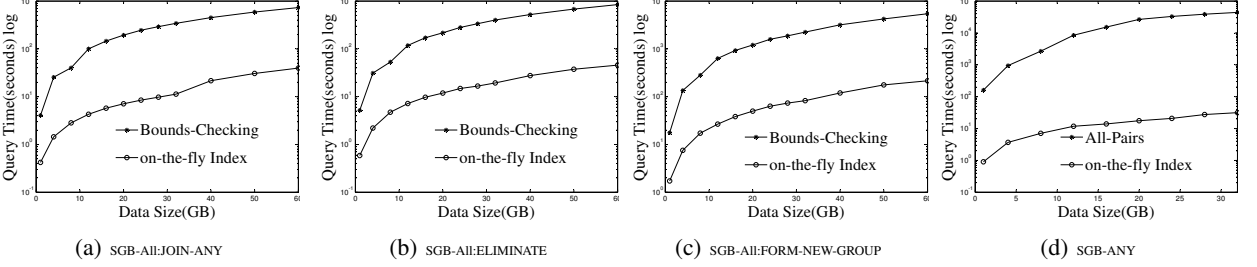


Fig. 10: The effect of increasing data size on the SGB-All variants and SGB-ANY

is that the Union-Find data structure efficiently finds and merges the candidate groups. Figure 9d verifies that, for all values of ϵ , the runtime performance of the *on-the-fly Index* method for SGB-Any is two orders of magnitude faster than the *All-Pairs* SGB-Any.

8.5 Speedup

Figure 10a, 10b and 10c give the performance and speedup of the *Bounds-Checking* and *on-the-fly Index* methods for large datasets with scale factor up to 60. The similarity threshold ϵ is fixed to 0.2. We do not show the results for the naive approach *All-Pairs* because its runtime increases quadratically as the data size increases. From Figure 10a, 10b and 10c, we observe that the runtime of the *Bounds-Checking* method increases as the number and size of groups increases. The *on-the-fly Index Bounds-Checking* method finds the sets *CandidateGroups* and *OverlapGroups* efficiently using the R-tree index, and the runtime of *on-the-fly Index Bounds-Checking* method increases steadily and is consistently lower than the *Bounds-Checking* methods. We observe that the speedup of the *on-the-fly Index Bounds-Checking* method is one order of magnitude better than that of *Bounds-Checking*.

Figure 10d gives the effect of varying the data size on the runtime of SGB-Any when ϵ is fixed to 0.2. The TPC-H scale factor (SF) ranges from 1 to 32. We observe that, as the data size increases, the runtime of the *All-Pairs* method increases quadratically, while the runtime of the *on-the-fly Index* method has a linear speedup. Moreover, the speedup results in the figure demonstrate that the *on-the-fly Index* method is approximately three orders of magnitude faster than *All-Pairs* SGB-Any as the data size increases.

8.6 Runtime Comparison with Clustering Algorithms

We compared the runtime of our SGB operators with three clustering algorithms, namely, *K-means* [9], *DBSCAN* [10], and *BIRCH* [12]. Specifically, we use the state-of-the-art implementation of *DBSCAN* with an R-tree from [22], the similarity threshold ϵ for both *DBSCAN* and SGB is set to 0.2, and the parameter K of *K-means* is set to 20 and 40, respectively. Figure 11 shows the proposed SGB operations significantly outperform *DBSCAN*, *BIRCH* and *K-means* by 1 to 3 order of magnitude on the real-world data respectively. The main reason is that the clustering algorithms scan the data more than once for convergence. On the contrary, SGB operations compute groups on-the-fly, and use group bound and a spatial index to reduce the overhead of distance computation with processed tuples. In addition, clustering algorithms have to read data from the database system making them slower than our built-in SGB operations.

8.7 Overhead of SGB

Figure 12 illustrates the effect of the various data sizes on the runtime of similarity-based groupings and traditional Group-By queries while varying the scale factor from 1G to 20G. The similarity threshold ϵ is fixed to 0.2. The semantics of the ON-OVERLAP clause plays a key role on the runtime of SGB-All. For instance, the JOIN-ANY variant achieves the best runtime among the SGB-All variants as it places overlapped elements into arbitrarily chosen groups. On the contrary, the FORM-NEW-GROUP incurs additional runtime cost while placing overlapped elements into new groups. The ELIMINATE semantics drops all overlapped elements causing the size of the output groups to shrink. Furthermore, the performance of

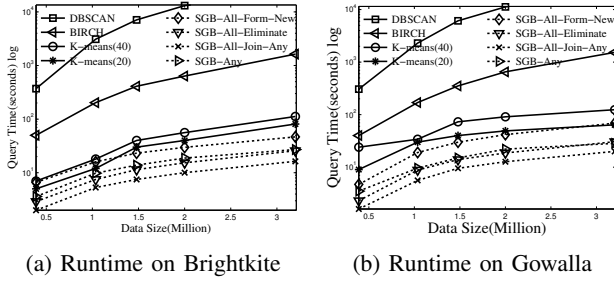


Fig. 11: SGB vs Clustering Algorithm

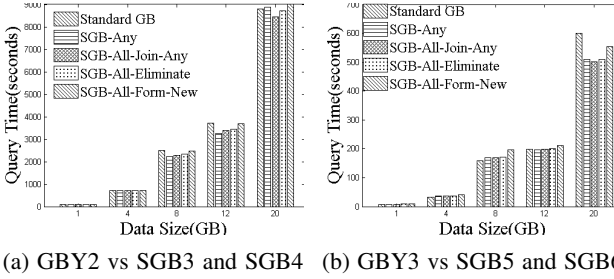


Fig. 12: The effect of the data size on SGB vs. SQL GBY.

traditional Group-by operator is comparable to the SGB-All and SGB-Any variants when using the *on-the-fly Index*. For instance, The SGB-All ON-OVERLAP JOIN-ANY shows better performance than that of traditional Group-By. The SGB-All ON-OVERLAP ELIMINATE, SGB-All ON-OVERLAP FORM-NEW and SGB-Any shows 15 percent, 40 percent and 20 percent overhead than the traditional Group-By, respectively.

9 CONCLUSION

In this paper, we address the problem of similarity-based grouping over multi-dimensional data. We define new similarity grouping operators with a variety of practical and useful semantics to handle overlap. We provide an extensible algorithmic framework to efficiently implement these operators inside a relational database management system under a variety of semantic flavors. The performance of SGB-All performs up to three orders of magnitude better than the naive *All-Pairs* grouping method. Moreover, the performance of the optimized SGB-Any performs more than three orders of magnitude better than the naive approach. Finally, the performance of the proposed SGB operators is comparable to that of standard relational Group-by.

REFERENCES

- [1] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques*. Morgan kaufmann, 2006.
- [2] Y. N. Silva, W. G. Aref, P.-A. Larson, S. S. Pearson, and M. H. Ali, "Similarity queries: their conceptual evaluation, transformations, and processing," *The VLDB Journal*, vol. 22, no. 3, pp. 395–420, 2013.
- [3] S. Adali, P. Bonatti, M. L. Sapino, and V. Subrahmanian, "A multi-similarity algebra," in *ACM Sigmod Record*, vol. 27, no. 2. ACM, 1998, pp. 402–413.
- [4] S. Atnafu, L. Brunie, and H. Kosch, "Similarity-based operators and query optimization for multimedia database systems," in *IDEAS*, 2001, pp. 346–355.

- [5] B. Braunmuller, M. Ester, H.-P. Kriegel, and J. Sander, "Multiple similarity queries: A basic dbms operation for mining in metric databases," *KDE*, vol. 13, no. 1, pp. 79–95, 2001.
- [6] J. Y. Chen and J. V. Carlis, "Similar_join: extending dbms with a bio-specific operator," in *SAC*, 2003, pp. 109–114.
- [7] H. L. Razente, M. C. N. Barioni, A. J. Traina, and C. Traina Jr, "Aggregate similarity queries in relevance feedback methods for content-based image retrieval," in *SAC*, 2008, pp. 869–874.
- [8] P. Berkhin, "A survey of clustering data mining techniques," in *Grouping multidimensional data*. Springer, 2006, pp. 25–71.
- [9] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *PAMI*, vol. 24, no. 7, pp. 881–892, 2002.
- [10] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," in *ACM SIGMOD Record*, vol. 25, no. 2, 1996, pp. 103–114.
- [11] S. Guha, R. Rastogi, and K. Shim, "Cure: an efficient clustering algorithm for large databases," in *SIGMOD*, vol. 27, no. 2. ACM, 1998, pp. 73–84.
- [12] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *KDD*, vol. 96, 1996, pp. 226–231.
- [13] E. Schallehn, K.-U. Sattler, and G. Saake, "Efficient similarity-based operations for data integration," *Data & Knowledge Engineering*, vol. 48, no. 3, pp. 361–387, 2004.
- [14] C. Zhang and Y. Huang, "Cluster by: a new sql extension for spatial data aggregation," in *GIS*, 2007, p. 53.
- [15] M. C. N. Barioni, H. Razente, A. Traina, and C. Traina Jr, "Siren: A similarity retrieval engine for complex data," *VLDB*, pp. 1155–1158, 2006.
- [16] D. Gulati, E. V. de Melo, R. M. Rangayyan, and R. C. Soares, "Postgresql-ie: An image-handling extension for postgresql," *Journal of digital imaging*, vol. 22, no. 2, pp. 149–165, 2009.
- [17] A. Guttman, *R-trees: A dynamic index structure for spatial searching*. ACM, 1984, vol. 14, no. 2.
- [18] M. De Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational geometry*. Springer, 2008.
- [19] R. E. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," *J. ACM*, vol. 31, no. 2, pp. 245–281, Mar. 1984.
- [20] "TPC-H Version 2.15.0." [Online]. Available: <http://www.tpc.org/tpch/>
- [21] E. Cho, S. A. Myers, and J. Leskovec, "Friendship and mobility: User movement in location-based social networks," in *Proceedings of the 17th ACM SIGKDD*. ACM, 2011, pp. 1082–1090.
- [22] E. Achtert, H.-P. Kriegel, E. Schubert, and A. Zimek, "Interactive data mining with 3d-parallel-coordinate-trees," ser. SIGMOD '13. ACM, 2013, pp. 1009–1012.
- [23] M. J. Atallah, "Computing the convex hull of line intersections," *J. Algorithms*, vol. 7, no. 2, pp. 285–288, 1986.

APPENDIX

We analyze the runtime of SGB-All and SGB-Any. Let n , k , $|G|$, $|G_c|$, $|G_v|$ be the data cardinality, the expected number of points per group, the number of existing groups, the size *CandidateGroups*, and the size of *OverlapGroups*, respectively, where $k \leq n$ and $|G| \leq n$ as each point can belong to only one group.

.1 SGB-All

The runtime for SGB-All is output-sensitive and is influenced by several factors e.g., the ON-OVERLAP options, and the runtimes of *FindCloseGroups* and *ProcessOverlap*. These factors vary with ϵ and with the data distribution. For instance, the number of Groups $|G|$ can vary from 1 to n depending on the value of ϵ . For example, when ϵ is very small, $|G| = n$. Next, we analyze the runtime complexity for *Bounds-Checking* and the *on-the-fly index*

for *Bounds-Checking* using the various ON-OVERLAP options.

SGB-All Join-Any. Refer to Procedure 4 *Bounds-Checking*. It finds the groups *CandidateGroups* by linearly testing all existing groups (Lines 4-6) to determine if point p_i can join Group g_j . Each test takes constant time. Thus, the runtime of ON-OVERLAP JOIN-ANY is bounded by the number of groups, i.e., $O(n |G|)$.

Refer to Procedure 5. *Groups_IX* is an *on-the-fly* R-tree that indexes the bounding rectangles of all existing groups. Given a new data point, say p_i , a window query of size 2ϵ on *Groups_IX* finds the groups *CandidateGroups* that p_i can join. Thus, the runtime for Procedure 5 (Line 4) is $O(\log |G|)$ and the overall runtime of ON-OVERLAP JOIN-ANY is $O(n \log |G|)$. When $|G| = n$ (the number of inputs tuples), the worst-case runtime of the *on-the-fly Index for Bounds-Checking* ON-OVERLAP JOIN-ANY is no better than $O(n \log n)$. In contrast, when $|G|$ is constant, e.g., 1, the best-case runtime is $O(n)$. Finally, the average-case runtime of the *on-the-fly index for Bounds-Checking* is $O(n \log |G|)$.

SGB-All Eliminate. The semantics of ON-OVERLAP ELIMINATE incurs additional $(k |G_v|)$ time while inspecting Set *OverlapGroups* to retrieve the subset that satisfies the similarity predicate (Lines 8-10) in Procedure 4 and (Lines 10-12) in Procedure 5). In addition, *ProcessEliminate* (Line 13) in Procedure 3 incurs additional cost of $|G_c|$ to update the bounds of the candidates groups after removing the overlapped points. Thus, the runtime of *Bounds-Checking* ON-OVERLAP ELIMINATE is $O(n (|G| + |G_c| + |G_v| k))$ while the runtime of *on-the-fly Index for Bounds-Checking* ON-OVERLAP ELIMINATE is $O(n (\log |G| + |G_c| + |G_v| k))$. Naturally, $k = n/|G|$, so the runtime of *on-the-fly Index for Bounds-Checking* ON-OVERLAP ELIMINATE is $O(n (\log |G| + |G_c| + n |G_v|/|G|))$. In the worst-case, $|G| = n$, $|G_c| = |G|$ and $|G_v|/|G| = \text{constant}$, and the corresponding runtime of *on-the-fly Index for Bounds-Checking* ON-OVERLAP ELIMINATE is $O(n^2)$. In contrast, the best-case runtime is $O(n)$ when the sizes $|G| = |G_v| = |G_c| = 1$. The average-case runtime is $O(n \log |G|)$ when the sizes of *OverlapGroups* $|G_v| \ll n$ and *CandidateGroups* $|G_c| \ll n$.

SGB-All FORM-NEW-GROUP. Procedures *ProcessNewGroup* and *ProcessOverlapNewGroup* insert the overlapped points into a temporary set S' . Upon finding all points in S' , SGB-All recursively performs a new round of FORM-NEW-GROUP while grouping the contents of S' until S' is empty. Let m be the recursion counter that is initially 0, and S'_m be the set S' at recursion stage m . Then, S'_0 is the input dataset where the size of S'_0 i.e., $|S'_0| = n$. The time cost for each round is $t_m = O(|S'_m| (O(\text{FindCloseGroupsALL}) + O(\text{ProcessOverlap})))$ that is $t_m = O(|S'_m| (|G^m| + |G_c^m| + |G_v^m| k^m))$, where $|G^m|$, $|G_c^m|$ and $|G_v^m|$ are the number of existing groups, *CandidateGroups*, and *OverlapGroups* at each round m , respectively. Thus, the overall runtime of SGB-All FORM-NEW-GROUP is the sum of t_m from recursion

depth 0 to DP , where t_m is the cost at Recursion Depth m . Then, the complexity of *Bounds-Checking* is $\sum_{m=0}^{DP} t_m = \sum_{m=0}^d O(|S'_m| (|G^m| + |G_c^m| + |G_v^m| k^m))$. Similarly, the time complexity of the *on-the-fly index* for *Bounds-Checking* is $\sum_{m=0}^d O(|S'_m| (\log |G^m| + |G_c^m| + |G_v^m| k^m))$. The best-case behavior of *Index Bounds-Checking* for FORM-NEW-GROUP occurs when set *OverlapGroups* is empty and the size of *CandidateGroups* is constant. Then, the best-case runtime is $O(n)$. In contrast, if the recursion depth is almost n , the worst-case runtime is $O(n^3)$. On average, the recursion counter $m = \text{constant} \ll n$ and $|S'_m| \ll n$, and the complexity is $O(m n \log(|G|))$.

The Convex Hull Test in Section 6.4 forms a convex hull for each group g_j to filter out the false-positive points. The expected size of the convex hull for one group g_j is h , where $h = \log k$ [23], where k is the expected number points in g_j . Refer to Procedure 6. It takes $O(\log h)$ to test if a point is inside the convex hull (Line 2). Moreover, given a point, say p_i , located outside the convex hull, it takes $O(\log h)$ to obtain the farthest point from p_i (Line 5). Thus, for a group of points, g_j , the time to test if p_i can join g_j is $O(\log h + \log h)$; that is $O(\log \log k)$. *ConvexHullTest* is performed for each group that passes the *PointInRectangle* test with $O(\log k)$ cost (using L_∞). Thus, the computation cost to extend Procedures 4 and 5 with *ConvexHullTest* is $O(n |G| \log k)$ for *Bounds-Checking* and $O(n \log |G| \log k)$ for the *on-the-fly Index* for *Bounds-Checking*. Finally, the average-case runtime of the *on-the-fly Index* for *Bounds-Checking* when using L_2 is $O(n \log |G| \log k)$. Notice that the actual running time is faster than the average-case because the convex hull test is executed only if a new point has passed the Group g_j 's rectangle test.

.2 SGB-Any

Refer to Procedure 8. For each new input point p_i , the window query returns the processed points that are within ϵ from p_i . Given a set of n points, the complexity of the window query is $O(n \log n)$. Moreover, Procedures *getGroups* and *MergeGroupsInsert* use Union-Find to keep track of new, existing, and merged groups. The amortized runtime of Union-Find for n points is $O(m' \alpha(n))$ [19], where m' is the total operations to build new groups, $m' = |G|$, $\alpha(n)$ is a very slowly growing function, and $\alpha(n) \leq 4$. Therefore, the average case of Union-Find running time is $O(n)$, where $m' \leq n$. Hence, the average-case runtime of SGB-Any using an *on-the-fly index* is $O(n \log n) + O(n)$, that is $O(n \log n)$. Also, using L_2 requires an additional step (*verifyPoints*) to filter out the points that do not satisfy the similarity predicate in *OverlapGroups* (Line 7) with a cost k' per point, where k' is the expected number of points within a window query. Consequently, the runtime cost of SGB-Any using L_2 is $O(n \log n + n k')$. k' is influenced by ϵ . Thus, the worst-case runtime when using L_2 is n^2 , when $k' \approx n$. If k' is constant, the average-case runtime is $O(n \log n)$. The average-case runtime of the *on-the-fly Index* for SGB-Any is $O(n \log n)$ for both L_∞ and L_2 .